

Chapter Five

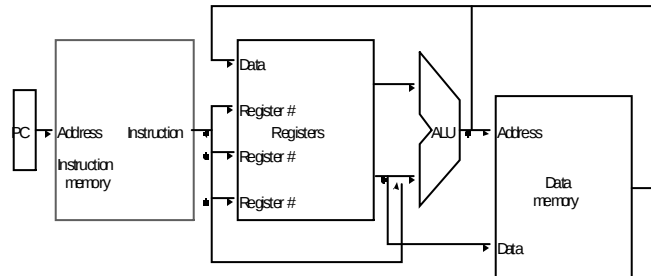
The Processor: Datapath and Control

The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
 - memory-reference instructions: **lw, sw**
 - arithmetic-logical instructions: **add, sub, and, or, slt**
 - control flow instructions: **beq, j**
- Generic Implementation:
 - use the program counter (PC) to supply instruction address
 - get the instruction from memory
 - read registers
 - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers
Why? memory-reference? arithmetic? control flow?

More Implementation Details

- **Abstract / Simplified View:**



Two types of functional units:

- elements that operate on data values (combinational)
- elements that contain state (sequential)

An unclocked state element

- **The set-reset latch**
 - output depends on present inputs and also on past inputs



Latches and Flip-flops

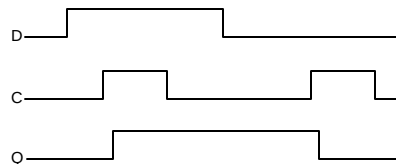
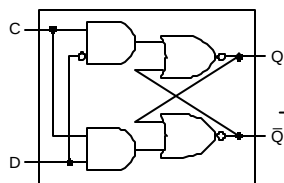
- Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge (edge-triggered methodology)

"logically true",
— could mean electrically low

A clocking methodology defines when signals can be read and written
— wouldn't want to read a signal at the same time it was being written

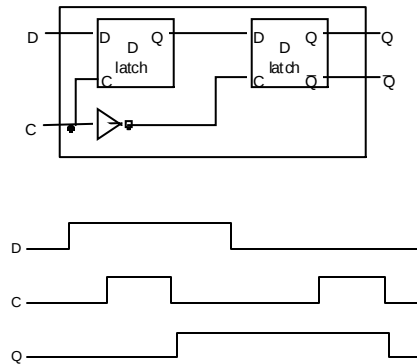
D-latch

- Two inputs:
 - the data value to be stored (D)
 - the clock signal (C) indicating when to read & store D
- Two outputs:
 - the value of the internal state (Q) and it's complement



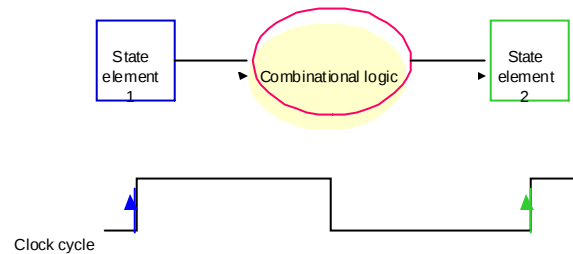
D flip-flop

- Output changes only on the clock edge



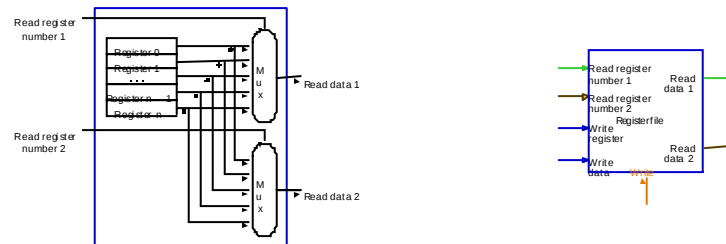
Our Implementation

- An edge triggered methodology
- Typical execution:
 - read contents of some state elements,
 - send values through some combinational logic
 - write results to one or more state elements



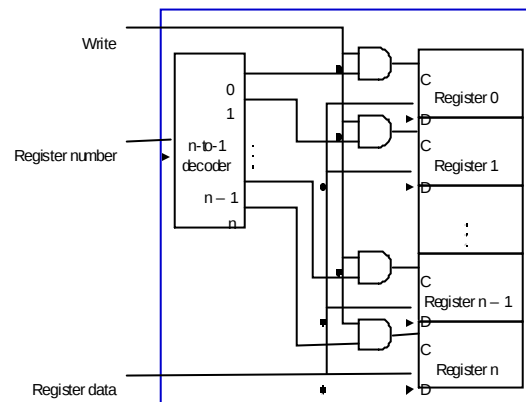
Register File

- Built using D flip-flops



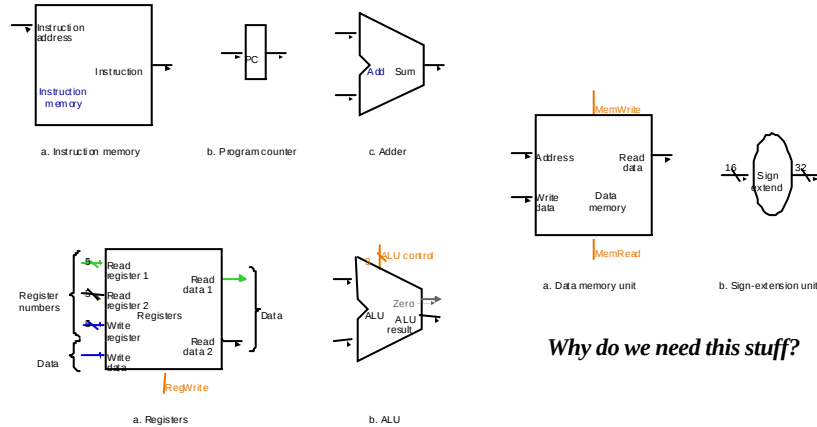
Register File

- Note: we still use the real clock to determine when to write



Simple Implementation

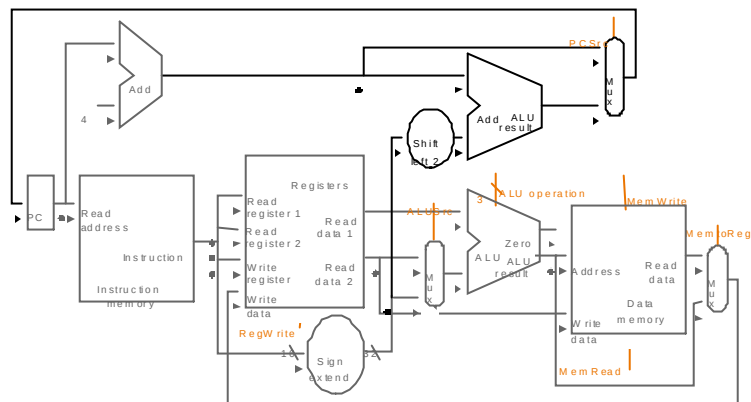
- Include the functional units we need for each instruction



Why do we need this stuff?

Building the Datapath

- Use multiplexers to stitch them together



Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexer inputs)
- Information comes from the 32 bits of the instruction
- Example:

add \$8, \$17, \$18

Instruction Format:

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

- ALU's operation based on instruction type and function code

Control

- e.g., what should the ALU do with this instruction
- Example: lw \$1, 100(\$2)

35	2	1	100
----	---	---	-----

op	rs	rt	16 bit offset
----	----	----	---------------

- ALU control input

000	AND
001	OR
010	add
110	subtract
111	set-on-less-than

- Why is the code for subtract 110 and not 011?

Control

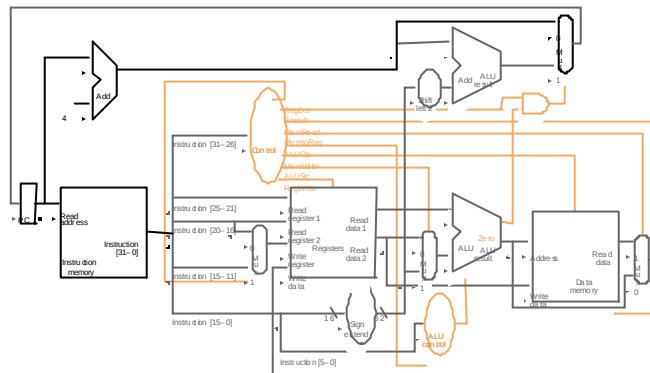
- Must describe hardware to compute 3-bit ALU control input
 - given instruction type
 - 00 = lw, sw
 - 01 = beq,
 - 11 = arithmetic
 - function code for arithmetic
- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

EE424 Spring 2003

15

Control



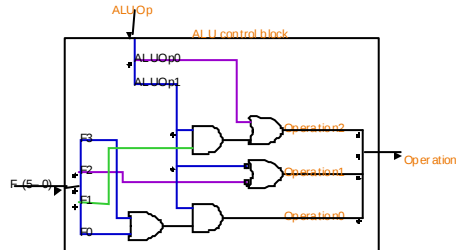
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

EE424 Spring 2003

16

Control (Part 1)

- Simple combinational logic (truth tables)



ALUOp		Funct field						Operation	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	010	lw/sw
0	1	X	X	X	X	X	X	110	branch
1	X	X	X	0	0	0	0	010	add
1	X	X	X	0	0	1	0	110	sub
1	X	X	X	0	1	0	0	000	and
1	X	X	X	0	1	0	1	001	or
1	X	X	X	1	0	1	0	111	slt

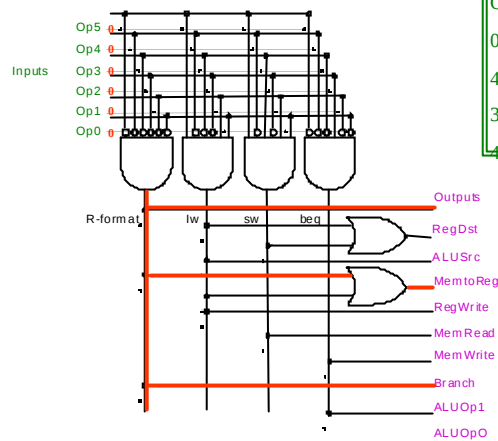
EE424 Spring 2003

17

Control

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- Simple combinational logic (truth tables)



From Figure A.19

Opcode [31:26]

0 000000 R-format inst

4 000100 branch inst.

35 100011 lw inst.

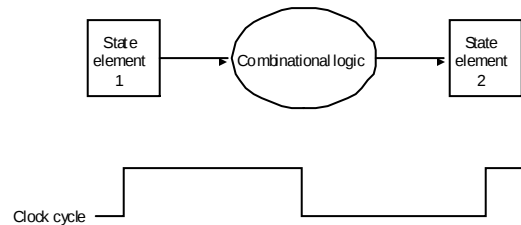
43 101011 sw inst.

EE424 Spring 2003

18

Our Simple Control Structure

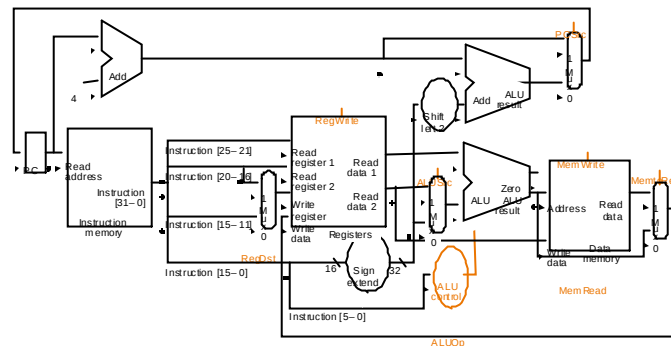
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
 - ALU might not produce “right answer” right away
 - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

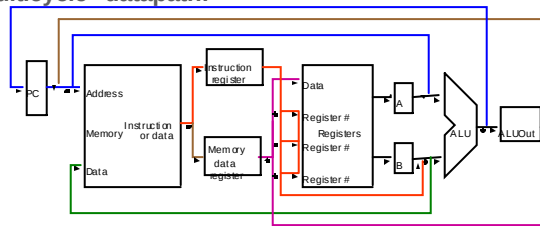
Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
 - memory (2ns), ALU and adders (2ns), register file access (1ns)



Where we are headed

- **Single Cycle Problems:**
 - what if we had a more complicated instruction like floating point?
 - wasteful of area
- **One Solution:**
 - use a “smaller” cycle time
 - have different instructions take different numbers of cycles
 - a “multicycle” datapath:



EE424 Spring 2003

21

Multicycle Approach

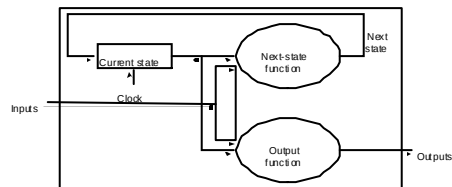
- **We will be reusing functional units**
 - ALU used to compute address and to increment PC
 - Memory used for instruction and data
- **Our control signals will not be determined solely by instruction**
 - e.g., what should the ALU do for a “subtract” instruction?
- **We'll use a finite state machine for control**

EE424 Spring 2003

22

Review: finite state machines

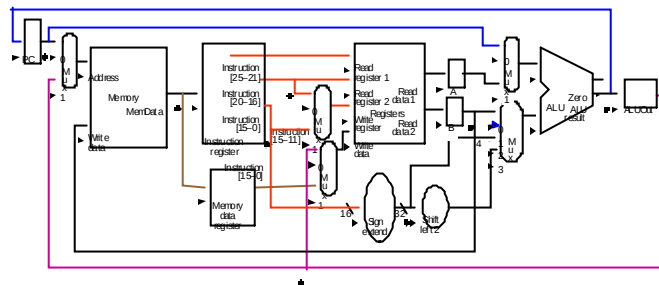
- **Finite state machines:**
 - a set of states and
 - next state function (determined by current state and the input)
 - output function (determined by current state and possibly input)



- We'll use a Moore machine (output based only on current state)

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
 - balance the amount of work to be done
 - restrict each cycle to use only one major functional unit
- At the end of a cycle
 - store values for use in later cycles (easiest thing to do)
 - introduce additional “internal” registers



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register .
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register -Transfer Language"

```
IR = Memory[PC];  
PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

- R-type:

```
ALUOut = A op B;
```

- Branch:

```
if (A==B) PC = ALUOut;
```

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR = Memory[ALUOut];  
or  
Memory[ALUOut] = B;
```

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Write-back step

- `Reg[IR[20-16]] = MDR;`

What about all the other instructions?

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A == B) then PC = ALUOut	PC = PC [31-28] (IR[25-0] << 2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Simple Questions

- How many cycles will it take to execute this code?

```

lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label1
add $t5, $t2, $t3
sw $t5, 8($t3)
Label:
...
```

#assume not

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of \$t2 and \$t3 takes place?

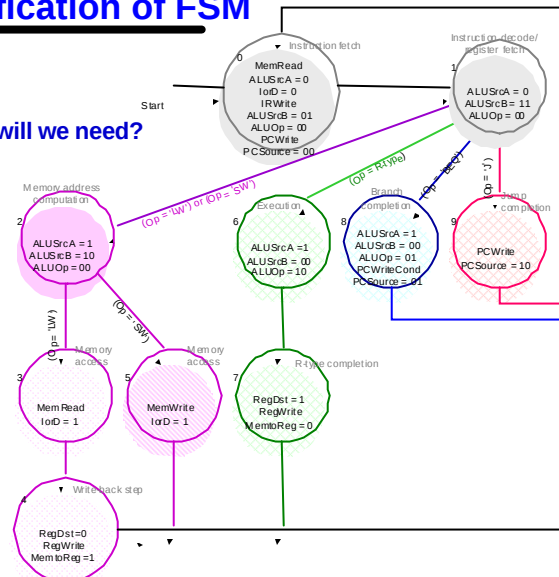


Implementing the Control

- Value of control signals is dependent upon:
 - what instruction is being executed
 - which step is being performed
- Use the information we've accumulated to specify a finite state machine
 - specify the finite state machine graphically, or
 - use microprogramming
- Implementation can be derived from specification

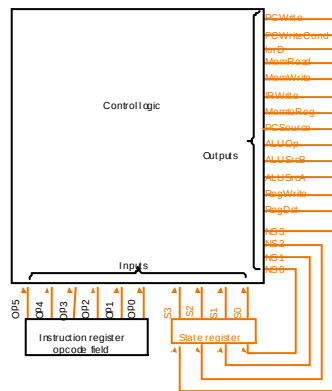
Graphical Specification of FSM

How many state bits will we need?



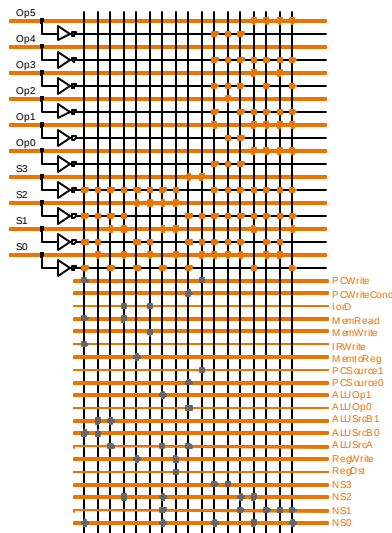
Finite State Machine for Control

- Implementation:



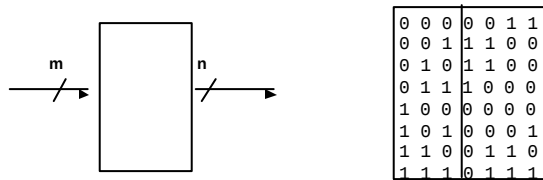
PLA Implementation

- If I picked a horizontal or vertical line could you explain it?



ROM Implementation

- ROM = "Read Only Memory"
 - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
 - if the address is m -bits, we can address 2^m entries in the ROM.
 - our outputs are the bits of data that the address points to.



m is the "height", and n is the "width"

ROM Implementation

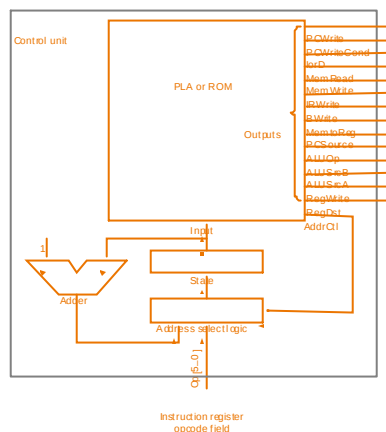
- How many inputs are there?
 - 6 bits for opcode, 4 bits for state = 10 address lines
(i.e., $2^{10} = 1024$ different addresses)
- How many outputs are there?
 - 16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is $2^{10} \times 20 = 20K$ bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
 - i.e., opcode is often ignored

ROM vs PLA

- Break up the table into two parts
 - 4 state bits tell you the 16 outputs, $2^4 \times 16$ bits of ROM
 - 10 bits tell you the 4 next state bits, $2^{10} \times 4$ bits of ROM
 - Total: 4.3K bits of ROM
- PLA is much smaller
 - can share product terms
 - only need entries that produce an active output
 - can take into account don't cares
- Size is $(\#inputs \times \#product\text{-terms}) + (\#outputs \times \#product\text{-terms})$
 For this example = $(10 \times 17) + (20 \times 17) = 460$ PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Another Implementation Style

- **Complex instructions:** the "next state" is often current state + 1



Details

Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

41

EE424 Spring 2003

Microprogramming

The diagram illustrates the architecture of a microprogrammed processor, divided into three main functional blocks: Control unit, Microcode memory, and Datapath.

- Control unit:** This block contains the logic for sequencing microinstructions. It includes:
 - Microprogram counter:** Receives an 'Input' and outputs an 'Address' to the Address select logic.
 - Address select logic:** Receives the 'Address' from the counter and outputs a 'Microcode field' to the Instruction register.
 - Instruction register:** Stores the current microinstruction, with its 'opcode field' outputting a 'Microcode field' back to the Address select logic.
- Microcode memory:** Receives the 'Microcode field' from the instruction register and provides multiple outputs to the Datapath:
 - PCWrite
 - PCWriteCond
 - PC
 - MemRead
 - MemWrite
 - IOWrite
 - IORead
 - MemtoReg
 - RegWrite
 - ALUSrc
 - ALUSrcB
 - ALUSrcC
 - RegWrite
 - RegDst
 - ALUSrcA
- Datapath:** Receives control signals from the Microcode memory and performs operations based on these signals.

- What are the “microinstructions” ?

42

EE424 Spring 2003

Microprogramming

- A specification methodology
 - appropriate if hundreds of opcodes, modes, cycles, etc.
 - signals specified symbolically using microinstructions

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshift	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- Will two implementations of the same architecture have the same microcode?
- What would a microassembler do?

Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshift	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and output the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, InpD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, InpD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, InpD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00, PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Maximally vs. Minimally Encoded

- **No encoding:**
 - 1 bit for each datapath operation
 - faster, requires more memory (logic)
 - used for Vax 780 — an astonishing 400K of memory!
- **Lots of encoding:**
 - send the microinstructions through logic to get control signals
 - uses less memory, slower
- **Historical context of CISC:**
 - Too much logic to put on a single chip with everything else
 - Use a ROM (or even RAM) to hold the microcode
 - It's easy to add new instructions

Microcode: Trade-offs

- **Distinction between specification and implementation is sometimes blurred**
- **Specification Advantages:**
 - Easy to design and write
 - Design architecture and microcode in parallel
- **Implementation (off-chip ROM) Advantages**
 - Easy to change since values are in memory
 - Can emulate other architectures
 - Can make use of internal registers
- **Implementation Disadvantages, SLOWER now that:**
 - Control is implemented on same chip as processor
 - ROM is no longer faster than RAM
 - No need to go back and make changes

The Big Picture

